APPLICATION

FOR

UNITED STATES LETTERS PATENT

TITLE: FILTERING CALLS IN SYSTEM AREA NETWORKS

APPLICANT: HEMAL V. SHAH and ANNIE FOONG

CERTIFICATE OF MAILING BY EXPRESS MAIL

Express Mail Label No EE647282645US

I hereby certify that this correspondence is being deposited with the United States Postal Service as Express Mail Post Office to Addressee with sufficient postage on the date indicated below and is addressed to the Commissioner for Patents, Washington, D.C. 20231.

January 22, 2001
Date of Deposit

Signature

Francisco Robles
Typed or Printed Name of Person Signing Certificate

FILTERING CALLS IN SYSTEM AREA NETWORKS

BACKGROUND

The invention relates to filtering calls in system area networks.

System area networks (SANs) provide network connectivity among nodes in server clusters. Network clients typically utilize Transmission Control Protocol/Internet Protocol (TCP/IP) to communicate with the application nodes. Application node operating systems are responsible for processing TCP/IP packets.

TCP/IP processing demand at the application nodes, however, can slow system operating speeds. To address this, TCP/IP processing functions can be offloaded to remote TCP/IP processing devices. Legacy applications may use remote procedure call (RPC) technology using non-standard protocols to off-load TCP/IP processing.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 illustrates a computer system.

FIG. 2 illustrates an application node.

FIG. 3 is a flowchart of a method according to the invention.

FIG. 4A illustrates a file descriptor.

FIG. 4B illustrates partitioning of file descriptors.

FIG. 5A illustrates a set of exemplary application calls and corresponding lightweight protocol messages.

FIG. 5B provides functional descriptions of exemplary lightweight protocol message types.

FIGS. 6A - 6S are psuedo-code for mapping application calls.

DETAILED DESCRIPTION

The computer system 10 of FIG. 1 includes network clients 12, a system area network (SAN) 14 and a SAN management node 22. The network clients 12 may exist, for example, either on a local area network (LAN) or a wide area network (WAN). The SAN 14 has one or more network nodes 16a … 16k, one or more proxy nodes 18a … 18k, and one or more application nodes 20a, 20b, 20c … 20k.

The network nodes 16a … 16k are platforms that can provide an interface between the network clients 12 and the

SAN 14. The network nodes 16a … 16k may be configured to perform load balancing across multiple proxy nodes 18a … 18k. The proxy nodes 18a … 18k are platforms that can provide various network services including network firewall functions, cache functions, network security functions, and load balancing logic. The proxy nodes 18a … 18k may also be configured to perform TCP/IP processing on behalf of the application nodes 20a, 20b, 20c … 20k. The application nodes 20a, 20b, 20c … 20k are platforms that function as hosts to various applications, such as a web service, mail service, or directory service. The application nodes 20a, 20b, 20c … 20k may, for example, include a computer or processor configured to accomplish the tasks described herein.

SAN channels 24 interconnect the various nodes. SAN channels 24 may be configured to connect a single network node 16a … 16k to multiple proxy nodes 18a … 18k, to connect a single proxy node 18a … 18k to multiple network nodes 16a … 16k and to multiple application nodes 20a, 20b, 20c … 20k, and to connect a single application node 20a, 20b, 20c … 20k to multiple proxy nodes 18a … 18k. The SAN channels 24 connect to ports at each node.

Network clients 12 utilize TCP/IP to communicate with proxy nodes 18a … 18k via network nodes 16a … 16k. A TCP/IP packet may enter the SAN 14 at a network node 16a and travel through a SAN channel 24 to a proxy node 18a. The proxy node 18a may translate the TCP/IP packet into a message based on a lightweight protocol. The term "lightweight protocol" refers to a protocol that has low operating system resource overhead requirements. Examples of lightweight protocols include Winsock-DP Protocol and Credit Request/Response Protocol. The lightweight protocol message may then travel through another SAN channel 24 to an application node 20a.

Data can also flow in the opposite direction, starting, for example, at the application node 20a as a lightweight protocol message. The lightweight protocol message travels through a SAN channel 24 to the proxy node 18a. The proxy node 18a translates the lightweight protocol data into one or more TCP/IP packets. The TCP/IP packets then travel from the proxy node 18a to a network node 16a through a SAN channel 24. The TCP/IP packets exit the SAN 14 through the network node 16a and are received by the network clients 12.

FIG. 2 shows an architectural view of an application node 20a based on an exemplary SAN hardware that uses a Virtual Interface (VI) Network Interface Card (NIC) 40. Legacy applications 30 traditionally utilize stream sockets application program interface (API) 32 for TCP/IP-based communication.

A stream socket filter 34 transparently intercepts application socket API calls and maps them to lightweight protocol messages communicated to proxy nodes 18a … 18k. The stream socket filter 34 provides a technique for applications in application nodes 20a, 20b, 20c … 20k to communicate with network clients 12, located external to the SAN 14, via the proxy nodes 18a … 18k and the network nodes 16a … 16k. The stream socket filter 34 is typically event-driven. A single lightweight protocol message sent or received by the stream socket filter 34 can serve more than one sockets API call. Thus, unnecessary round-trips may be minimized for calls that do not generate any network events. The stream socket filter 34 may reside between an application and a legacy network stack. The stream socket filter 34 may be implemented as a dynamically loadable library module (where supported by the operating system), or as a statically linked library (where recompilation of the source is possible).

The SAN Transport 36, Virtual Interface Provider Library (VIPL) 38, and the Network Interface Card (NIC) 40 are standard components that allow the application node 20a to perform lightweight protocol-based communications.

In legacy applications, sockets are software endpoints used for communications between application nodes 20a, 20b, 20c … 20k and network clients 12. Sockets may be opened either actively or passively on an associated file descriptor (socket).

Applications 30 issue requests for actions to take place in the form of calls issued on a file descriptor. As shown in FIG. 3, the stream socket filter 34 may intercept 50 an application's call. The stream socket filter then determines 52 whether communication with a proxy node 18a is needed 52 by examining the call issued on a given file descriptor and by examining the file descriptor. If the stream socket filter 34 determines that communication with a proxy node 18a is not needed, then the stream socket filter 34 processes 54 the call locally and returns an appropriate response to the caller. If the stream socket filter 34 determines that communication with the proxy node 18a is required, then for an outgoing message (i.e., a message received from an application 30), the stream socket

filter 34 translates 55 the message to a lightweight protocol message and sends 56 the message to a proxy node 18a. If the message is incoming (i.e., received from a proxy node 18a), the stream socket filter 34 receives 60 the lightweight protocol message. The stream socket filter 34 then determines 57 whether further communication is needed with a proxy node 18a. If further communication is required, the stream socket filter 34 repeats the above process. If further communication is not needed with a proxy node 18a, the stream socket filter 34 returns 58 an appropriate response to the caller.

The stream socket filter 34 determines whether a network event should be generated (block 52) by considering the call issued and the file descriptor. As illustrated in FIG. 4A, the file descriptor 80 can be, for example, a sixteen-bit data structure. The file descriptor may be assigned by the application node's operating system 26a.

As shown in FIG. 4B, the range 90 of available file descriptors includes all valid combinations of data based on a particular data structure. For the sixteen-bit data structure 80 of FIG. 4A, the available file descriptors range from all zeros (binary 0) to all ones (binary 65,535). In order for legacy applications to preserve host operating system descriptors on the application nodes 20a,

20b, 20c … 20k, the stream socket filter 34 partitions the 16-bit file descriptor range 90 into traditional file descriptors 92, which are assigned by the operating system; and transport file descriptors 94, which are assigned by the proxy nodes 18a … 18k. Each transport file descriptor 94 corresponds to a unique flow identifier (flow id) used by the proxy node 18a in labeling the corresponding TCP flow.

Traditional file descriptors that are assigned by the operating system lie in the range between zero and FD_SETSIZE-1, which typically has the value of 1023. File descriptors between the value of FD_SETSIZE-1 and 65535 are typically available for use by the proxy node 18a to communicate with the stream socket filter 34.

A socket() call in an application typically returns a file descriptor 80 whose value is provided by the application node operating system 26a, 26b, 26c … 26k. This file descriptor may be bound to a well-known port for listening on a connection. If this happens, the file descriptor is then categorized as a service file descriptor 98. Service file descriptors 98 may be used to distinguish between different service sessions between an application node 20a and a proxy node 18a. The operating system may also assign file descriptors known as mapped file

descriptors 99. Any other file descriptors in the OS-assigned range that are not service file descriptors 98 or mapped file descriptors 99 may typically be used for file input/output or network input/output related functions, usually unrelated to the proxy node 18a or SAN transport 36 functions.

The stream socket filter 34 may use transport file descriptors 94 for both actively and passively opened stream sockets. For passively opened TCP-related sockets, a flow identifier ("flow id") supplied by a proxy node 18a may be returned by the accept() call as the file descriptor to be used by the application 30. The file descriptor returned is actually a transport file descriptor 94 taking on the value of the flow id associated with that particular flow. Some applications (e.g. File Transfer Protocol servers) make a connect() call to a network client 12 to actively open a socket on the application node 20a. Since the application node operating system 26a typically generates the file descriptor prior to connection establishment, the file descriptor typically needs to be mapped to a transport file descriptor 94 when the connection is finally established. The application may use the operating system 26a assigned mapped file descriptors 99, whereas the stream socket filter 34 may use the

corresponding transport file descriptors 99 for communication.

The stream socket filter 34 recognizes which of the categories (system, service, mapped or transport) a particular file descriptor falls under. Based on that categorization and based on the particular call issued, the stream socket filter 34 determines whether a communication with a proxy node 18a is necessary.

As shown in FIG. 5A, the left hand column lists a set of calls that an exemplary legacy application 30 might issue. The right hand column lists corresponding lightweight protocol messages that the stream socket filter 34 might issue in response to those calls. Not all application calls require network events. Calls that do not require network events may be processed locally by the application node's operating system 26a.

An application 30 on an application node 20a typically starts a service with a socket() call. An endpoint is then initialized. If an application 30 issues a bind() call followed by a listen() call, the stream socket filter 34 notes the service file descriptor 98 and then sends a JOIN_SERVICE message containing the service file descriptor 98 to the proxy node 18a indicating that the application 30

is ready to provide application services. The application 30 then waits for a network client's 12 request via a select() or an accept() call. The stream socket filter 34 intercepts the select() or accept() call and waits for the arrival of a CONNECTION_REQUEST message from the proxy node 18a. The CONNECTION_REQUEST message typically arrives with a flow id assigned by the proxy node 18a, which is then returned to the application 30 in response to the accept() call. The application 30 may then use the returned flow id as the transport file descriptor 99 for subsequent reading and writing of data.

The stream socket filter 34 may map read and write calls from the application 30 onto DATA messages. If an application 30 finishes its data transfer on a particular transport file descriptor 94, it typically invokes a close() call, which the stream socket filter 34 will translate to a CLOSE_CONNECTION message that is sent to the proxy node 18a. When the application 30 is ready to shutdown its services, it invokes a close() call on a service file descriptor 98, which the stream socket filter 34 recognizes, triggering a LEAVE_SERVICE message to be sent to the proxy node 18a, and terminating the services.

Not all application calls generate communication messages. Calls that do not require generating lightweight protocol messages (e.g., socket () and bind () calls) may be processed locally.

FIG. 5B provides descriptions of typical lightweight protocol messages that may be generated in response to application calls.

FIGS. 6A - 6S provide exemplary pseudo-code describing typical responses that a stream socket filter 34 may make for exemplary application calls. Each of these figures describes responses to a particular application call issued. Other sockets API calls, particularly setsockopt() and getsockopt(), may primarily set and get the intended behavior of socket operation for the application nodes 20a, 20b, 20c, … 20k. These settings may be kept in global state variables, which may or may not have a meaningful impact on the socket-filtered calls, since a reliable SAN Transport may be used in place of TCP. Where necessary, such information may also be relayed to the proxy nodes 18a … 18k, as they may be responsible for the TCP connection to the network clients 12, on behalf of the application nodes 20a, 20b, 20c … 20k. For data transfer related calls, the pseudo-codes typically assume synchronous operations and fully opened sockets.

Systems implementing the techniques described herein are also capable of implementing techniques for error handling, parameter validation, address checking, as well as other standard techniques.

Systems implementing the foregoing techniques may realize faster SAN 14 operating speeds and improved system flexibility. The techniques described herein may alleviate operating system legacy networking protocol stack on servers bottlenecking for inter-process communication (IPC) in a SAN. Operating system related inefficiencies incurred in network protocol processing, such as user/kernel transitions, context switches, interrupt processing, data copies, software multiplexing, and reliability semantics may be minimized, and may result in an increase in both CPU efficiency and overall network throughput. With TCP/IP processing offloaded to proxy nodes 18a … 18k, a lightweight protocol based on SAN Transport 36 may be used in the SAN 14 and may reduce processing overheads on application servers. The stream socket filter 34 may enable legacy applications that use socket-based networking API to work in a SAN 14 and/or network with non-legacy communication protocols, in conjunction with proxy nodes 18a … 18k.

Various features of the system may be implemented in hardware, software or a combination of hardware and software. For example, some aspects of the system can be implemented in computer programs executing on programmable computers. Each program can be implemented in a high level procedural or object-oriented programming language to communicate with a computer system. Furthermore, each such computer program can be stored on a storage medium, such as read-only-memory (ROM) readable by a general or special purpose programmable computer, for configuring and operating the computer when the storage medium is read by the computer to perform the functions described above.

Other implementations are within the scope of the following claims.